EFFICIENT INFERENCE OF CONVOLUTIONAL NEURAL NETWORKS ON GENERAL PURPOSE HARDWARE USING WEIGHT REPETITION

BY

ROHIT AGRAWAL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Assistant Professor Christopher W. Fletcher

ABSTRACT

Deep Neural Networks (DNNs) have begun to permeate all corners of electronic society due to their high accuracy and machine efficiency per operation. Recent work has shown how weights within and across DNN filters have large degrees of repetition due to the pigeonhole principle and modern weight quantization schemes, and that this weight repetition can be harnessed improve DNN inference efficiency in an accelerator/ASIC context. This thesis develops new techniques so that weight repetition leads to an efficiency gain on general-purpose and programmable SIMD-based architectures such as CPUs equipped with vector extensions. We show how to write high-performance software that does not require hardware modifications and can cope with the irregularity introduced by weight repetition schemes. Overall, our highly parallel software kernel achieves upto 1.51× speedup in runtime of inference over state-of-the-art baseline.

To my family, for their love and support.
To Vanishka, The Queen of the Universe.

ACKNOWLEDGMENTS

I joined University of Illinois, Urbana-Champaign in Fall 2017 and it has been a fun roller coaster ride since then, all of which led me to this moment. This ride wouldn't have been what it was without the guidance of my advisor, Professor Christopher W. Fletcher. My goal before starting master's was to try my hands on research. After working for almost 2 years with him, I can happily *"declare victory"*, as Chris would. I would also like to give a special mention to Christopher J. Hughes from Intel Labs for invaluable discussions and guidance.

In my opinion, quality research is not possible without having people around with whom one can freely discuss, develop and kill ideas. I was fortunate to have Hadi Aghsari Moghaddam, Jiyong Yu, Jose Rodrigo Sanchez Vicarte, Kartik Venkatraman Hegde, Mohamad El Hajj and Muhammad Haris Mughees as my colleagues and friends. I couldn't have asked for a more friendly, motivated and funny group of people to work with. History lessons and movie nights with Hadi, Cricket and politics discussions with Haris, Jiyong's collection of memes, road trip with Jose and late night brainstorming sessions and Japan trip with Kartik deserve special mention.

I would not have made it through this degree without the support of friends I made at UIUC. Abhinav Kohar, Abhishek Srivastava, Aditi Arvind, Alok Ranjan, Dhruv Agarwal, Dipali Ranjan, Malabika Koley, Priyasmita Ghosh, Shivank Mishra and Sidharatha Satapathy have been a great source of fun, entertainment and motivation throughout my stay at UIUC. I will always cherish the laughter and happiness that we shared in both good and bad times during our stay at UIUC. I would also like to thank Harsh Singh, Satyanshu Kumar and Saumya Shah, my friends since undergrad, who helped a great amount from the time of graduate school applications to final transition to UIUC.

Finally, to my parents, I owe a great deal to them for their endless love and affection. They taught me the value of simplicity and self-belief which will stay with me forever. I would also like to thank my brother and my sister-in-law for their support.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Deep Neaural Networks (DNNs) have become a buzzword in recent years due to their profound impact on world's technological, social and economic activities in areas as diverse as healthcare, entertainment, scientific research and driving. At present, DNNs are proven to be extremely effective in domains such as computer vision, speech and text as they have outperformed almost all previous algorithms [1, 2, 3, 4, 5]. As a result, they are being extensively deployed in cloud, mobile and IoT setting as a major workload for these tasks [6, 7, 8]. Given their current trajectory, DNNs can be expected to comprise a large chuck of workloads and consume most of the world's compute resources.

DNNs are typically used in two phases: first is the training phase where it is expected to learn about the problem, e.g., image classification [9, 10, 11, 12], by training on a huge amount of data. Second is the inference phase where it is queried on real-life data to solve problems. Training is a rather long and computational intensive process and is usually done offline in data centers whereas inference requires efficient online processing in either cloud or edge devices. Given that training is a one time process whereas inference is employed across several devices and is done very frequently, it is only natural for researchers to try and improve its efficiency.

Of particular interest are Convolutional Neural Networks (CNNs), which currently hold the state-of-the-art results in many vision tasks such as image/temporal action recognition [9, 13], and scene generation [14]. At their core, CNNs are massively parallel high-dimensional dot products between inputs and sliding CNN filters (containing weights). In the past, there have been significant advancements in improving CNN efficiency by exploiting sparsity (presence of zero)[15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]) in CNN filters and inputs. There have also been major advancements in reducing the bit-precision of weights and inputs (quantization) [26, 27, 28, 15, 29]) and different data-movement pattern (dataflow) [22, 21, 30, 17, 31]) to improve CNN efficiency. Although these directions have had major breakthroughs because they are general principles that apply across various domains, deployments and devices needed for inference, they quickly saturated and their limitations became apparent. It is critical that we develop new techniques in this vein, which can likewise improve DNN inference efficiency in its multitude of settings.

A promising new direction is reducing redundant and superfluous work by exploiting repetition in weights present in CNN filters. Consider figure 1.1a where a simple 1-dimensional convolution is shown. It slides a filter of 3 weights $\{a, b, a\}$ across inputs $\{x, y, z, k, l...\}$, computing a dot product at each step which forms the output (i.e., $\{ax+by+az, ay+bz+ak, ...\}$).

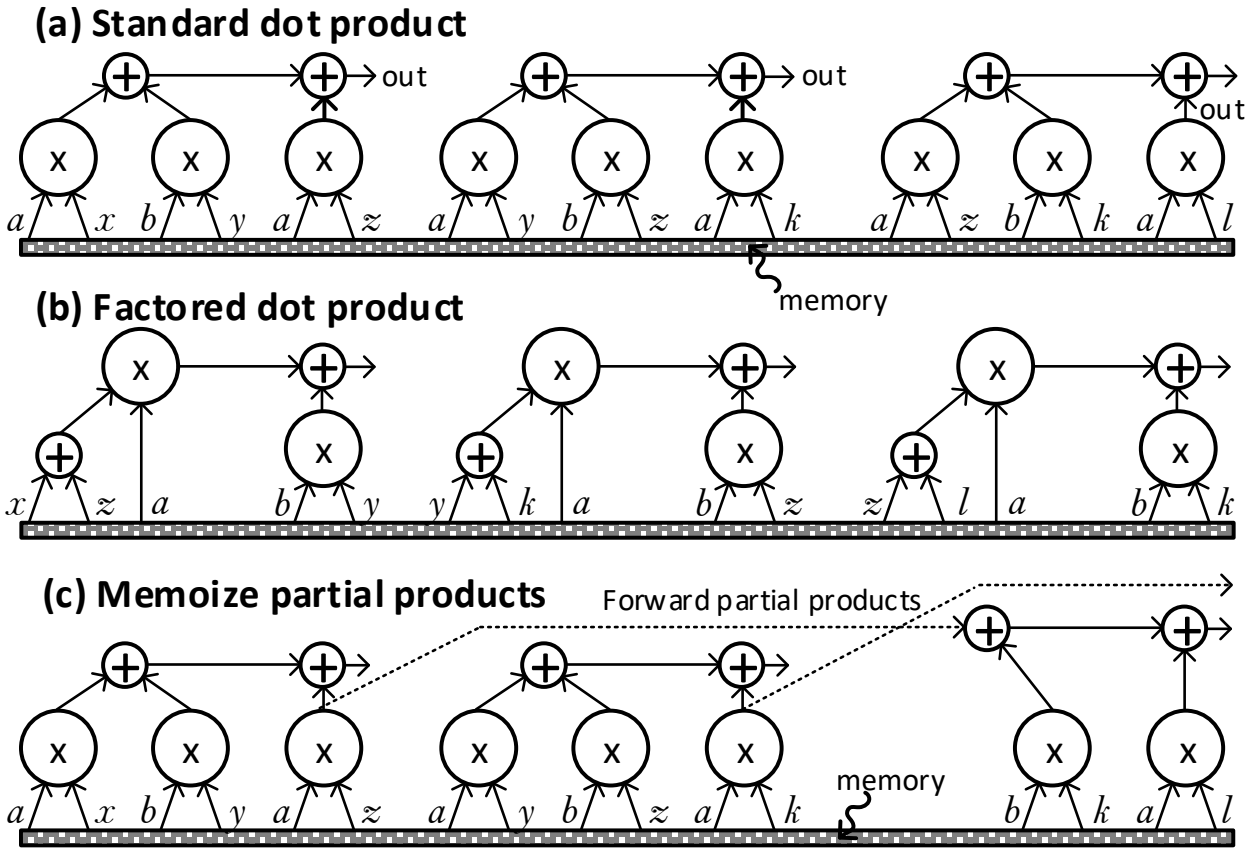

Figure 1.1: Standard (a) and different optimized (b, c) 1D convolutions that take advantage of repeated weight a. Arrows out of the grey bars indicate input/filter memory reads. Our goal is to reduce memory reads, multiplications and additions while obtaining the same result.

We refer to elements in the input as the activations, elements in the filter as the weights and the number of weights in the filter as the filters size (3 in this case). When evaluated on hardware, this computation requires reading each input and weight from memory, and performing a multiply-accumulate (MACC) operation on that input-weight pair. In this case, each output is calculated using 3 input reads, 3 weight reads, 3 multiplications and 2 additions.

Figure 1.1b and 1.1c provides two alternate but functionally equivalent ways of performing this dot product. 1b considers the fact that weight a appears twice in the filter and can be used to remove its product with activation z while simultaneously reducing the extra memory read for a. We call this *Dot Product Factorization*. Performing the convolution in this fashion requires 3 input reads, 2 weight reads and 2 multiplications and 2 additions which is equivalent to saving 33% multiplies and 16% memory reads. 1c exploits repetition of weight a to reuse the partial result generated during first dot-product ($a.z$) in subsequent outputs when the filter slides by two positions, saving 33% multiplies and memory reads.

We call this *Memoization of Partial Product*.

Both of the above mentioned techniques rely on the fact that a weight in the filter is repeated and can be used to improve efficiency by saving computations and memory reads. More importantly, the phenomenon of weight repetition complements existing techniques of sparsity [15, 16, 20, 22, 23, 21] and quantization [26, 27, 28, 15, 29] as sparsity is just a special case of repetition of zero weights and repetition increases with more aggressive quantization due to pigeonhole principle. The above examples used a dummy filter with only 3 weights whereas filters in modern CNNs can have thousands of weights and they may be seeing weight repetition already. For example, representing each weight in 8-bits means there can be ≤ 256 unique weights where filter size can easily be in thousands. Moreover, weight quantization techniques like INQ [27] (17 unique weights) and TTQ [28] (3 unique weights) enable a great degree of repetition in modern CNN filters.

Previous work [32] has shown the abundance of weight repetition in modern network and various ways they can be exploited in an accelerator setting to get a net performance boost while simultaneously compressing the model. This generalizes a popular and rich line of work focusing on ASIC design to exploit sparsity in CNNs [21, 25, 19, 20, 22, 23]. However, as the name suggests, ASICs are generally optimized for a particular set of CNNs and they can hardly keep up with the extremely fast evolving networks. This is also one of the main reason general purpose and programmable devices like CPUs and GPUs are being used to perform inference in data centers. As [7] suggests, Facebook currently relies heavily on CPUs for inference and both CPUs and GPUs for training so improving CNNs performance on CPUs becomes extremely worthy.

However, exploiting repetition to get a net performance improvement is a challenge for several reasons. First, CNN filters can have extremely irregular pattern of repetition which results in irregular data access pattern which completely undermines CPU's ability to hide latency by techniques like caching, prefetching etc. Second, tracking the repetition pattern adds metadata which results in increased pressure on both memory (for storage) and bandwidth (to move metadata around). Third, fixed L1 cache doesn't allow more than a certain amount of data to be kept near the compute resource which restricts the amount of repetition that can be exploited. Fourth, its extremely challenging in general to design an algorithm and corresponding software which doesn't have any overhead of its own and can completely utilize the hardware to get theoretical speedup.

This work addresses these challenges by designing a hardware-aware software kernel which can perform CNN inference on a SIMD style general purpose CPU. This is done in two parts. First, we develop a novel algorithm called *Activation Group Reuse* which piggybacks on *Dot Product Factorization* to compute CNN dot product in an optimized fashion while

simultaneously compressing the model. The compression rate is competitive to that given by aggressive weight quantization schemes [27, 28], and gives an added ability to exploit weight repetition. Second, we carefully map the algorithm on a general purpose CPU using SIMD instructions to perform CNN inference.

Contributions. To summarize, this work makes the following contributions:

1. To the best of our knowledge, this is the first work to assess the possibility of exploiting weight repetition on general purpose CPUs.

2. We design a weight repetition aware software kernel and make several performance optimizations to efficiently map it on CPUs.

3. We evaluate the performance of our kernel on several state-of-the-art CNNs for image classification task against MKL_DNN which is a industry standard software kernel for CNN inference.

4. We present this work as a proof of concept for weight repetition being a general way to perform efficient DNN inference on general purpose hardware while simultaneously compressing the model.

CHAPTER 2: BACKGROUND

2.1 CNN BACKGROUND

CNNs consist of multiple operations like convolution, pooling, non-linearity [33] and multilayer perceptron with convolution being the dominant operation in terms of compute. Each of these operation is called a CNN layer and multiple such layers are stacked together in a pipeline fashion to form the entire network. A convolution operation in CNNs consist of sliding window convolution between $W \times H \times C$ input and $R \times S \times C$ filters to generate $(W - R + 1) \times (H - S + 1)$ output. This is a dot-product operation of inputs and filters for each spatial position in the output. There can be K such filters in a convolutional layer to form K output channels. This is visualized in figure 2.1. We will omit $'\times'$ from dimension for brevity when possible, for example, $W \times H \times C \rightarrow WHC$

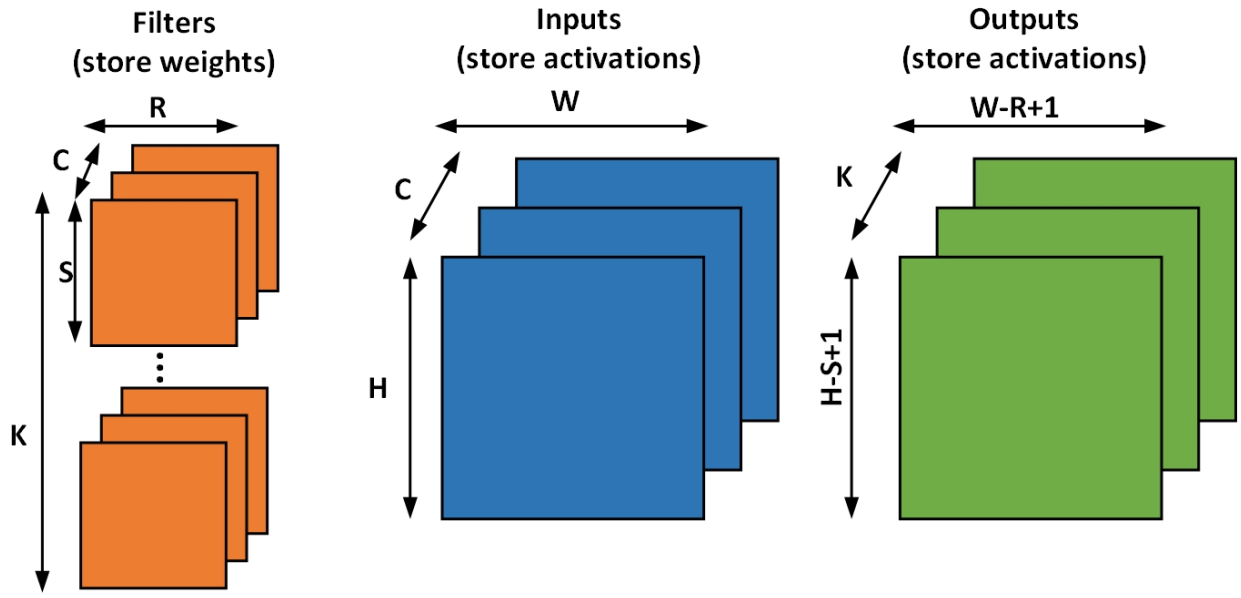

Figure 2.1: CNN parameters per convolutional layer

CNN Inference. As with prior work [19, 21, 20], this paper focuses on CNN inference which is usually performed online. As non-convolutional layers do not contribute much to the overall computations [34], we focus on improving the efficiency of convolutional layers only. Mathematically, inference for convolutional layers without bias terms and unit stride

can be written as follow.

$$O[(k,x,y)] = \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} F[(k,c,r,s)] \times I[(c,x+r,y+s)] \tag{2.1}$$

$$0 \leq k \leq K, 0 \leq x \leq W-R+1, 0 \leq y \leq H-S+1$$

where O, I and F are outputs (activations), input (activations) and filters (weights), respectively. Since the layers are stacked up in a pipeline fashion, the output of one layer becomes the input for next layer. As is the case with other work targeting inference [23], we assume a batch size of one.

2.2 WEIGHT REPETITION IN MODERN CNNS

We make a key observation that while the overall filter volume have been relatively constant over time, the number of unique weights have decreased consistently. This is largely the consequence of successful approaches to reduce overall size of the CNN model [15, 28, 27, 16, 35]. This has been achieved primarily by two mechanisms both referred to as *weight quantization* schemes. The first approach targets to reduce the bit-precision of weights which reduces the overall model size and the cost of arithmetic operation [29]. The second one is to use a small number of high precision weights [15, 28, 27] which also compresses the model but can achieve higher accuracy that simply reducing precision.

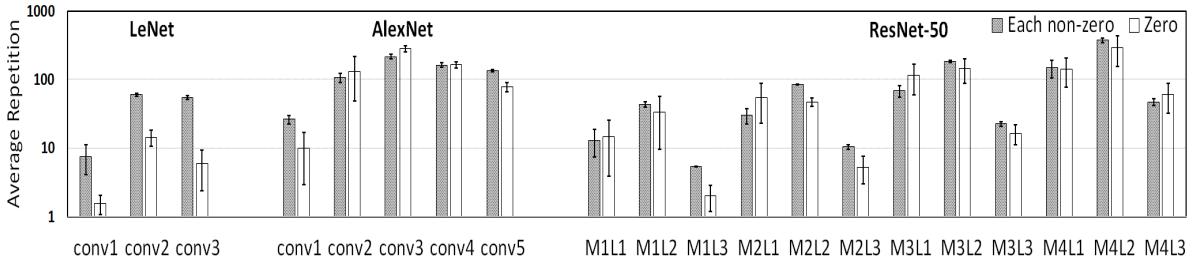

Figure 2.2: Weight repetition per filter, averaged across all filters, for select layers in a LeNet like [36] CNN, AlexNet [9] and ResNet-50 [12]. All networks are trained with INQ [27]. LeNet was trained on CIFAR-10 [37] and AlexNet/ResNet on ImageNet [38]. MxLy stands for "module x, layer y". In the case of ResNet, we show one instance of each module, where repetition is averaged across filters in the layer. Note that the error bars represent the standard deviation of weight repetition in each layer.

A complementary line of work shows that it is possible to dramatically reduce the number of unique weights, while maintaining state-of-the-art accuracy, by decoupling the number of unique weights from the numerical precision of the weights [29, 28, 27]. Figure 2.2 shows the amount of repetition for three modern CNNs trained with a scheme called Incremental Network Quantization (INQ) [27]. INQ constraints the model to have only 17 unique weights (16 non-zero and one zero). It shows a LeNet-like [36] CNN trained on Cifar-10 [37] and Alexnet [9] and ResNet-50 [12] trained on Imagenet [38] which achieved 80.16%, 57.39% and 74.81% top-1 accuracy respectively.

Figure 2.2 shows the average repetition of each zero and non-zero weight within each filter. We see that repetition is widespread in modern CNNs in not only zero weights but in non-zero weights as well. More importantly, the repetition count for non-zero weights seldom drops below 10 which indicates that the combined repetition of all non-zero weights can dominate over pure sparsity. The key takeaway point here is that there is a large untapped potential opportunity to exploit weight repetition.

CHAPTER 3: EXPLOITING WEIGHT REPETITION

We now discuss the key mechanisms by which we can exploit weight repetition to improve efficiency both in terms of cycles and energy. We first discuss *dot product factorization* which exploits repetition in the RSC volume of a single filter to save multiplications. We then generalize this idea to exploit repetition within and across filters. i.e., throughout the RSCK dimension using *activation group reuse*. Lastly, we also present a third type of reuse that we do not exploit but it can be used in future work.

3.1 DOT PRODUCT FACTORIZATION

Given each dot product in the CNN (an RSC filter volume MACCed with an RSC sub-region of inputs), our goal is reduce the amount of computations required to compute the dot product. Given repeated weights, this can be achieved by factorizing the common weights in the dot product, as shown in figure 1.1b. The inputs that are multiplied to the same weight are grouped and summed locally and only that sum is multiplied with the common weight. We refer to this group of activations grouped locally an *activation group*. To summarize:

1. Each activation group correspond to one unique weight in the filter.
2. The number of activation groups is the same as the number of unique weights in the filter
3. The size of an activation group is same as the amount of repetition present in the filter for its corresponding unique weight.

We can now rewrite Equation 2.1 to include dot product factorization as follows

$$O[(k,x,y)] = \sum_{i=0}^{U} \Big(F[wiT[k,i]] \times \sum_{j=0}^{gsz(k,i)-1} I[iiT[(k,i,j)]] \Big) \tag{3.1}$$

O, F and I are outputs, filters and inputs from equation 2.1. gsz(k,i) indicates the size of the size of the i^{th} activation group in the k^{th} filter and U represents the number of unique weights in the layer. Note that each CNN layer can have unequal number of unique weights due to irregular distribution. This means that the activation group size for a unique weight can be zero for a filter.

Activations group are spread out irregularly within RSC region of inputs which makes it difficult to access them regularly. Thus, we need an indirection table to know the inputs

belonging to same activation group, i.e., the inputs that share a common weight. We call this an *input indirection table* or `iiT`. `iiT` stores the offset corresponding to each input in an activation group so that they can be read from the input buffer in activation group order. That is, $\texttt{iiT}[(k,i,0)]...\texttt{iiT}[(k,i,gsz(k,i)-1)]$ represents offset in the input space corresponding to activations in i^{th} activation group for filter k.

Correspondingly, we also need to determine which unique weight should be multiplied to each activation group. We store this information in a separate table called *weight indirection table* or simply `wiT`. $\texttt{wiT}[(k,i)]$ points to a weight that should be multiplied with i^{th} activation group in k^{th} filter. Note that since weights and thus repetition pattern is frozen after training, both these tables can be populated offline by one time processing after training. They also completely replace the original filter buffer as these two tables along with all the unique weights are sufficient to know everything about the filters.

Savings. The primary benefit of dot product factorization is the reduced number of multiplications per dot product. More specifically, we need to perform the same number of multiplications as there are number of unique weights in the filter regardless the size of the activation groups. From figure 2.2, this can give us savings from 5× to 373× in multiplications.

An important special case is zero weights, i.e., when $\texttt{wiT[(k,i)]} = 0$. In that case, the activation group becomes irrelevant and the accumulation and multiplication is skipped.

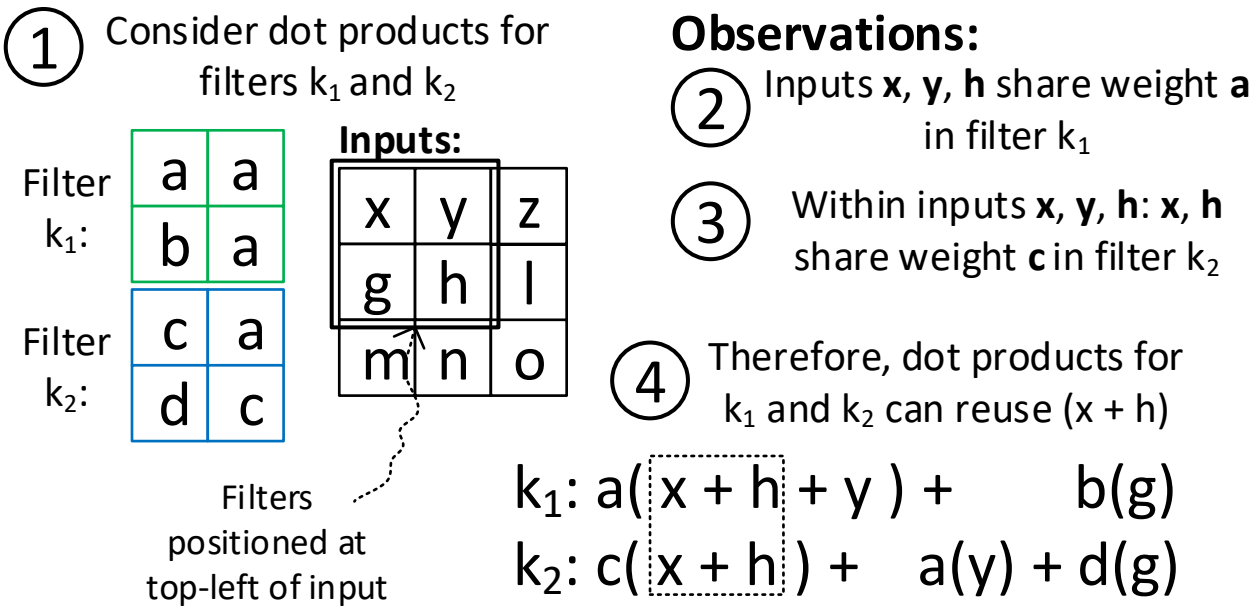

Figure 3.1: Activation group reuse example for G=2.

3.2 ACTIVATION GROUP REUSE

Dot product factorization reduces multiplications by exploiting weight repetition within an RSC volume of filter. We can generalize this idea to simultaneously exploit repetition across filters using a *activation group reuse*. The key idea is to exploit the overlap between repetition pattern, i.e., activation groups of two or more filters. In figure 3.1, activations $x + h + y$ form an activation group for filter k_1. Within this activation group, filter k_2 has a *sub-activation group* which is $x + h$. The intersection of these two $(x + h)$ can be reused across the two filters.

Formally, we can build the sub-activation groups for filter k_2, within i^{th} activation group of filter k_1, as follows. First we build the activation groups for k_1

$$A(k_1, i) = \{\texttt{iiT}[(k_1, i, j)] : j \in [0, gsz(k_1, i)]\}$$

Then, we ca sub-activation groups for filter k_2 by taking set intersections. That is, for $i' = 0, ..., U - 1$, the i'^{th} sub-activation group for k_2 is given by:

$$A(k_1, i) \cap A(k_2, i')$$

We can generalize this scheme to find overlap across G filters. When $G = 1$, we have vanilla dot product factorization (3.1). The discussion above is for $G = 2$. For $G > 2$, we can recursively form set intersections between filters k_g and $k_{(g+1)}$, for $g = 1, ...G - 1$. That is, once sub-activation groups for filter k_2 are formed, we can look for "sub-sub" activation groups within a filter k_3 and so on. Formally, suppose we have a g^{th} level activation group T_g for filter k_g. To find the $(g + 1)^{th}$ level activation groups for filter h_{g+1} within T_g, we calculate $T_g \cap A(k_{g+1}, i')$ for $i' = 0, ...U - 1$, which is analogous to how intersections were formed for $G = 2$ case.

Savings. Activation group reuse can bring significant improvements in two ways:

1. *Reduced input buffer reads and arithmetic operations:* From figure 3.1, we can eliminate the buffer reads and additions for reused sub-expressions like $x + h$. The scheme simultaneously saves multiplies as done in vanilla dot product factorization.

2. *Compressed input indirection table* `iiT`: Since we do not need to re-read the sub, sub-sub etc. activation groups for filters $k_2, ..., k_g$, we can reduce the size of the input indirection table `iiT` by an $O(G)$ factor.

How prevalent is activation group reuse? Activation group reuse is only possible when there are overlaps between the activation groups of two or more filters. If there are no

overlaps, we cannot form the compound sub-activation group expressions that can be reused across the filters. These overlaps are likely to occur when the filter size $R * S * C$ is larger than U^G. For example, for $(R, S, C) = (3, 3, 256)$ and $U = 8$, we expect to see overlaps between filter groups upto size $G = 3$ filters.

We experimentally found that the networks retrained with INQ [27] ($U = 17$) and TTQ [28] ($U == 3$) can enable $G = 2$ to 3 and $G = 6$ to 7 respectively for a majority of ResNet-50 layers. Note that these schemes can simultaneously achieve competitive classification accuracy relative to large U schemes.

Indirection table sorting. To reduce the size of the indirection tables and to simplify their traversal, we sort entries in the input and weight indirection tables such that reading the input indirections sequentially looks up the input buffer in activation group-order. The weight indirection table is read in the same order. Note that because sorting is a function of weight repetitions, it can be performed offline.

Importantly, the sorted order implies that each weight in the unique weight buffer need only be read out once per activation group, and that the weight indirection table need only store the index of the weight in the unique weight buffer instead of the actual weight value. Specifically, the next entry in the weight buffer is read whenever an activation group finishes and the weight buffer need only store U entries.

As mentioned in section 3.1, we dont store indirection table entries that are associated with the zero weight. To skip zeros, we sort the zero weight to the last position and encode a filter done message in the existing table.

3.3 PARTIAL PRODUCT REUSE

We make the following additional observation. While dot product factorization looks for repetitions in each RSC dimensional filter, it is also possible to exploit repetitions across filters, within the same input channel. That is, across the RSK dimensions for each input channel. This idea is shown for 1D convolution in Figure 1.1c. In CNNs, for each input channel C, if $w = F[(k_1, c, r_1, s_1)] = F[(k_2, c, r_2, s_2)]$ and $(k_1, r_1, s_1) \neq (k_2, r_2, s_2)$, partial products formed with weight w can be reused across the filters, for the same spatial position, and as the filters slide. We do not exploit this form of computation reuse further in this work, as it is not directly compatible with the prior two techniques.

CHAPTER 4: PERFORMANCE OPTIMIZATION TECHNIQUES

In this section, we will describe the optimization techniques which enable a net performance gain of the UCNN kernel on hardware. We start with a strawman CNN loopnest which gives a functionally correct output but is inefficient when deployed in a real-world setting. We then make changes to the loopnest to get UCNN kernel which supports weight repetition as discussed in chapter 3. Finally, we elaborate several performance optimizations that we do on the kernel with their motivation, advantages and potential shortcomings.

4.1 CNN LOOP NEST

As discussed in section 2.1, CNN computations are sliding window convolution of a RSC dimensional filter with a WHC dimensional input, where, RS and WH are spatial dimensions of filter and inputs respectively and C is the number of channels. There can be K number of such filters resulting in K number of output channels for inference on one image. This can be described in a nested loop around the core MACC operation of the dot product, one for each unique dimension described above. Since, these MACCs are commutative, there can be multiple ways the dot product can be performed corresponding to a permutation of the nested loop.

One such permutation is described in algorithm 4.1. Inputs, filters and outputs are stored in row-major format, i.e., $KCSR$ for filters, CHW for inputs and KHW for outputs. It performs an RS dot product for each of the spatial position in the output spatial plane to finish one output channel. It then iterates over all K filters to generate the entire set of output activations before processing the next image from a batch size of N. Since one output is completely baked before moving on to the next one, this loopnest is called *Output Stationary* [?]. As explained in section 2.1, we will assume a batch size of 1.

4.2 UCNN KERNEL: WEIGHT REPETITION AWARE LOOP NEST

The loopnest described in 4.1 accesses all the data types, viz., inputs, outputs and filters in an ordered row major fashion. Since the repetition pattern of a particular weight in the filter can be arbitrary, it cannot be traversed in a regular manner as before. We introduce two tables: *input indirection table* `(iiT)` and *weight indirection table* `(wiT)` to handle the irregularity and support *dot product factorization* as described in section 3.1. Each of the table has $R * S * C$ entries, one for each weight in the filter and they are traversed sequen-

Algorithm 4.1: CNN Loop Nest

```
1  input[C][H][W];
2  filter[K][C][S][R];
3  output[K][H-S+1][W-R+1];
4  for n: 1 → N do
5  |  for k: 1 → K do
6  |  |  for h: 1 → H do
7  |  |  |  for w: 1→ W do
8  |  |  |  |  psum = 0;
9  |  |  |  |  for r: 1 → R do
10 |  |  |  |  |  for s: 1→ S do
11 |  |  |  |  |  |  for c: 1→ C do
12 |  |  |  |  |  |  |  psum = input[c][h + s − 1][w + r − 1] ∗ filter[k][c][s][r];
13 |  |  |  |  |  |  end
14 |  |  |  |  |  end
15 |  |  |  |  end
16 |  |  |  |  output[k][h][w] = psum;
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

tially to get the actual weight and corresponding input through indirection. To support *activation group reuse*, we hierarchically sort the indirections to align sub-activation groups with activation groups as described in section 3.2. This changes the strawman loopnest to the one shown in algorithm 4.2.

Algorithm 4.2 shows UCNN kernel for $G = 2$. We use AG_g to represent an activation group at level g where $g = 0$ always represents outermost activation group and $g = G - 1$ represents innermost activation group. The algorithm traverses the tables to read inputs through indirections until it reaches a change in weight which means the end of an activation group. A change in weight for any value of g indicates the end of activation group at that level and all levels below it.

For $G = 2$, there can be 4 scenarios. First, when neither AG_0 nor AG_1 is complete. Given large enough repetition window, the innermost activation groups will be sufficiently large in size so that this becomes the common case. It consists of only accumulations of inputs into innermost activation group (AG_1). This is denoted by statement 24 in algorithm 4.2. Second, when AG_1 finishes but AG_0 doesn't. This requires performing a MACC for AG_1 with corresponding weight from $\texttt{wiT}_1$ and accumulating its parent activation group (AG_0 in this case). Third, when both AG_0 and AG_1 are complete. This requires accumulating AG_0

first and then performing MACCs for both AG_0 and AG_1 with corresponding weights from $\texttt{wiT}_0$ and $\texttt{wiT}_1$. Fourth, when AG_0 finishes but AG_1 doesn't. This is not possible because of the the way we sort the indirection tables as described in section 3.2. Note that each of the above case corresponds to one branch of the $if - else$ tree and only one of them will be taken at a time. This prevents us from doing any unnecessary work. Finally, after the entire table is traversed, the accumulated partial sums ($PSUM_0$ and $PSUM_1$) for both filters are stored back to memory.

Algorithm 4.2: UCNN Kernel for G=2

```
1  input[C][H][W];
2  iiT[K/G][C * S * R];
3  wiT_0[K/G][C * S * R];
4  wiT_1[K/G][C * S * R];
5  output[K][H − S + 1][W − R + 1];
6  for k: 1 → K/G do
7      for h: 1 → H do
8          for w: 1→ W do
9              for i: 1 → R*S*C do
10                 AG_0 = AG_1 = 0;
11                 PSUM_0 = PSUM_1 = 0;
12                 indirection = iiT[k][i]
13                 ip = input[indirection]
14                 if wiT_0[k][i] ≠ wiT_0[k][i − 1] then
15                     AG_0+ = AG_1;
16                     PSUM_1+ = AG_1 * wiT_1[k][i − 1];
17                     PSUM_0+ = AG_0 * wiT_0[k][i − 1];
18                     AG_0 = AG_1 = 0;
19                 else if wiT_1[k][i] ≠ wiT_1[k][i − 1] then
20                     AG_0+ = AG_1;
21                     PSUM_1+ = AG_1 * wiT_1[k][i − 1];
22                     AG_1 = 0;
23                 else
24                     AG_1+ = ip;
25                 end
26             end
27             output[k*G+0][h][w] = PSUM_0;
28             output[k*G+1][h][w] = PSUM_1;
29         end
30     end
31 end
```

Lack of Parallelism. The output stationary loopnest described above is weight repeti-

tion aware and avoids unnecessary multiplications as much as possible. However, it doesn't take advantage of any inherent parallelism present in CNNs apart from simultaneous compute with activation group reuse. For example, the loopnest can be parallelized for each set of output activations (at statement 6) or for each spatial position (at statement 7 and 8) or both. Parallelizing at 6 would mean that multiple output activations are computed simultaneously for each input giving *input reuse* whereas parallelizing at 7 or 8 would mean computing multiple spatial output in a channel together giving *weight reuse*.

Unutilized Spatial Locality. Modern CPUs are extremely good at hiding latency of regular data access by employing caches, prefetchers etc. which allow them to make full use of spatial locality. This generally means that the more spatial locality is present in data access, the better its performance will be. The kernel described above has very good spatial locality in the indirection table reads but can potentially have very poor locality in input access due to random indirections. This completely undermines the ability of modern CPUs to fetch the next set of data correctly that it *guesses* will be accessed by subsequent operations. Even though a cacheline of input will be fetched for every input load, there is no guarantee that the entire cacheline's data will be accessed before it gets evicted due to conflicts which can lead to severe performance loss.

4.3 SPATIAL VECTORIZATION

Modern CPUs support Single Instruction Multiple Data (SIMD) style computations using extended ISA. For example, Intel introduced Advanced Vector Extensions (AVX) in 2008 to provide direct access of the supported SIMD instructions to programmers. We use these instructions to spatially vectorize the UCNN loopnest in W dimension by vector width V_w and exploit filter reuse. This requires performing vector loads instead of scalar loads for inputs (at 13 in algorithm 4.2) and vectorized ADD and FMA for activation groups.

The vectorized UCNN kernel is shown in algorithm 4.3. Note that we now have to increment the spatial position in W dimension by V_w after each loop iteration. Spatial vectorization gives three key benefits. First, since both inputs and outputs are stored in row major fashion, vectorizing in W dimension allows contiguous access of inputs as opposed to random scalar access. This helps a great deal in utilizing the bandwidth provided at various level of memory hierarchy to fetch useful data as it is guaranteed that the entire fetched vector of inputs will be consumed immediately after it is available. Second, it provides a first degree filter reuse by computing multiple spatial output in an output channel in SIMD fashion. Third, it amortizes the cost of indirection table reads as a single indirection is now being used to fetch a vector of inputs instead of a scalar input. This gives a second degree

filter reuse.

Algorithm 4.3: Vectorized UCNN Kernel for G=2

```
1  input[C][H][W];
2  iiT[K/G][C * S * R];
3  wiT_0[K/G][C * S * R];
4  wiT_1[K/G][C * S * R];
5  output[K][H − S + 1][W − R + 1];
6  for k: 1 → K/G do
7      for h: 1 → H do
8          for w: 1→ W; w+= V_w do
9              for i: 1 → R*S*C do
10                 # AGs and PSUMs are vector
11                 AG_0 = AG_1 = 0;
12                 PSUM_0 = PSUM_1 = 0;
13                 indirection = iiT[k][i]
14                 # Vector load of inputs
15                 ip = vload(input, indirection)
16                 if wiT_0[k][i] ≠ wiT_0[k][i − 1] then
17                     AG_0 = vadd(AG_0, AG_1)
18                     PSUM_1 = vfma(AG_1, wiT_1[k][i-1], PSUM_1)
19                     PSUM_0 = vfma(AG_0, wiT_0[k][i-1], PSUM_0)
20                     AG_0 = AG_1 = setzero();
21                 else if wiT_1[k][i] ≠ wiT_1[k][i − 1] then
22                     AG_0 = vadd(AG_0, AG_1)
23                     PSUM_1 = vfma(AG_1, wiT_1[k][i-1], PSUM_1)
24                     AG_1 = setzero();
25                 else
26                     AG_1 = vadd(AG_1, ip)
27                 end
28             end
29             vstore(output[k*G+0][h][w], PSUM_0);
30             vstore(output[k*G+1][h][w], PSUM_1);
31         end
32     end
33 end
```

Increased working set size. Vectorized code gives tremendous speedup over scalar code for a massively parallel application like CNN. However, it increases the working set size by vector width, putting significant pressure on L1 cache which can lead to increased capacity misses and resulting in sub-optimal bandwidth utilization. In an ideal scenario, vector loads and arithmetic can be efficiently pipelined to achieve theoretical throughput

of the hardware. However, achieving this throughput realistically is challenging for several reasons. For example, one would expect the above described UCNN kernel to work very well since it mostly does vector loads and adds in the common case. Careful observation however, reveals that a kernel with G filters in a set of indirection table will have to go through G branches between vector load of inputs and the common case of add.

4.4 FAST PATH: ISOLATING COMMON CASE

The presence of conditional branches between a vector load of inputs and subsequent accumulation in the common case proves to be a major bottleneck in performance. Even with extremely accurate branch predictors in modern CPUs, it is difficult to hide the effects of G branches specially if we want to compute multiple output channels together with large G. We realize that the kernel breaks out of the common case at the end of the innermost activation group and G conditional branches to detect the end is inefficient. So we separate out the common case with the rest of the kernel by pre-computing the size of the innermost activation groups. This can be done offline after populating `iiT` and `wiT`.

We introduce another data structure called *Common Case Table* (`ccT`) to store the size of each innermost activation group in the set of indirection tables. This allows us to know the exact number of indirection reads we need to perform before we break out of the innermost activation group. Given a filter of size $R*S*C$ and U number of unique weights, average size of innermost activation group for G filters in a set of indirection tables will be $(RSC)/U^G$. This means a total of U^G entries in ccT on average, one for each innermost activation group. This is a relatively small storage and scalar read overhead compared to the savings from skipping G conditional branches.

The introduction of `ccT` changes the UCNN kernel to the one shown in algorithm 4.4. Before traversing the indirection tables, we read an entry from ccT to get the size of current innermost activation group. This helps in trivial load and accumulate of inputs without encountering any *if-else* tree. We call this common case *Fast Path* and the *if-else* tree the *Slow Path*. The fast path consist of only four operations: one conditional branch, scalar `iiT` load, vector input loads and vector input adds, the last two of which can be performed using a single memory mode vector add instruction. We additionally unroll the fast path by a factor called `IAG_THRESHOLD` which stands for *Innermost Activation Group Threshold*. This allows us to further speed up the fast path by amortizing the cost of conditional branch. In the unrolled fast path, we read `IAG_THRESHOLD` number of `iiT` entries and perform the same number of vector input loads before accumulating them using vector adds. In the case where the size of innermost activation group goes below the specified threshold, we

fall back to original *Medium Fast Path* where we perform one scalar load and one memory mode vector add at a time until the entire inner activation group is accumulated. Note that `IAG_THRESHOLD` is an optimization parameter which can be tuned according to U and G so that we stay in fast path most of the time. Splitting the UCNN kernel into the above mentioned parts greatly simplifies and optimizes the kernel. With a properly tuned `IAG_THRESHOLD`, we stay in fast path most of the time and encounter the expensive *if-else* ladder of slow path only when required. This serves two purposes. First, it avoids any superfluous work of reading `wiT` entries and executing the branch condition. Second, the unrolled fast path enables less frequent transition between vector and scalar operations in the kernel which allows infrequent frequency transition due to dynamic frequency scaling.

RAW dependency. Although fast path reduces some of the performance bottlenecks by a significant amount, a careful look into it reveals that it itself is sub-optimal. Since it accumulates a single innermost activation group, the vector adds have Read-After-Write (RAW) dependencies among them which prevents from achieving the maximum possibe throughput of functional units.

4.5 SPATIAL UNROLLING: BREAKING THE DEPENDENCY CHAIN

We spatially unroll the UCNN kernel discussed in 4.4 to hide the RAW dependencies of vector adds in the fast path. This requires unrolling the kernel in H dimension in addition to spatial vectorization in W dimension. Note that since W and H are identical from CNN computation perspective, we can choose to perform spatial vectorization in any of the two and spatially unroll the other depending on the data layout of inputs and outputs.

Each of the vector operation in the kernel is now unrolled in H dimension. The unrolling factor depends on the latency of vector adds on the machine. For example, Intel Skylake-X with AVX-512 intrinsics has a vector add latency of 4 cycles and has 2 add ports. So an unrolling factor of 8 is required to completely hide the RAW dependencies.

Assuming the vector loads in fast path hit in L1, spatial unrolling allows to us to compute at the peak throughput of ALUs. This also amortizes the cost of indirection table reads similar to spatial vectorization since multiple spatial outputs are computed for a single input indirection. However, it increases the working set size by the unrolling factor which exerts storage pressure on the L1. Also, since there are limited number of vector registers available per core, it is likely that unrolling by a large amount will cause a spill into memory. This will essentially turn an almost free register access to costly memory access. So, it becomes necessary to carefully control the unrolling factor according to the spatial dimensions of input which is discussed in chapter 5.

Algorithm 4.4: Vectorized UCNN Kernel for G=2 with Fast, Medium Fast and Slow Path

```
1  input[C][H][W];
2  iiT[K/G][C * S * R];
3  wiT_0[K/G][C * S * R];
4  wiT_1[K/G][C * S * R];
5  std :: vector  ccT[K/G];
6  output[K][H − S + 1][W − R + 1];
7  for k: 1 → K/G do
8      for h: 1 → H do
9          for w: 1→ W; w+= V_w do
10             for i: 1 → R*S*C do
11                 # AGs and PSUMs are vector
12                 AG_0 = AG_1 = 0;
13                 PSUM_0 = PSUM_1 = 0;
14                 IAG_size = ccT[k].pop();
15                 # Fast Path
16                 while IAG_size ≥IAG_THRESHOLD do
17                     # Unrolled IAG_THRESHOLD times
18                     indirection = iiT[k][i];
19                     ip = vload(input, indirection);
20                     AG_1 = vadd(AG_1, ip);
21                     i++;
22                     IAG_size -= IAG_THRESHOLD;
23                 end
24                 # Medium Fast Path
25                 while IAG_size > 0 do
26                     indirection = iiT[k][i];
27                     ip = vload(input, indirection);
28                     AG_1 = vadd(AG_1, ip);
29                     i++;
30                     IAG_size −;
31                 end
32                 # Slow Path
33                 if wiT_0[k][i] ≠ wiT_0[k][i − 1] then
34                     AG_0 = vadd(AG_0, AG_1);
35                     PSUM_1 = vfma(AG_1, wiT_1[k][i-1], PSUM_1);
36                     PSUM_0 = vfma(AG_0, wiT_0[k][i-1], PSUM_0);
37                     AG_0 = AG_1 = setzero();
38                 else
39                     AG_0 = vadd(AG_0, AG_1);
40                     PSUM_1 = vfma(AG_1, wiT_1[k][i-1], PSUM_1);
41                     AG_1 = setzero();
42                 end
43             end
44             vstore(output[k*G+0][h][w], PSUM_0);
45             vstore(output[k*G+1][h][w], PSUM_1);
46         end
47     end
48 end
```

4.6 SPLIT LOADS AND MODIFIED MEMORY LAYOUT

The UCNN kernel described in 4.5 perform some number of vector load from input memory for each entry in the indirection table. Since these indirections can point to anywhere in the input memory, a vector of inputs from that location might not necessarily be cacheline aligned. This can result in the vector being spanned across two cachelines and both of them will have to be loaded in order to perform subsequent operations. This is called the problem of *Split Load* and it can potentially half the available bandwidth from memory to functional units. This can lead to significant performance degradation specially if the load requests are being satisfied from lower levels in the memory hierarchy.

Since we vectorize our computations in W dimension, split loads happen because of indirections pointing to random locations in input memory in that dimension which aren't cacheline aligned. Considering $R \times S$ as spatial dimension of the filter where R is the number of columns and S is the number of rows, we categorize the filters according to the value of R to understand more about the problem.

4.6.1 $1 \times s$ Filters

Most modern CNNs like ResNet-50, MobileNet have filters with only one columnn or in other words, their spatial dimension if of the form $1 \times s$ where s is the number of rows. This means that there is only one indirection in W dimension per channel (corresponding to r=0). For our kernel where we vectorize in W dimension, split loads for $1 \times s$ type filters can be removed by padding the inputs so that each spatial row size is a multiple of vector width. This requires a small storage overhead of padding.

4.6.2 Non $1 \times s$ Filters

In cases where there are more than 1 spatial column in the filter, there can be multiple indirections in W dimension. This complicates the problem as trivial padding won't fix it for indirections with $r \neq 0$. This can be handled in two steps.

First, we pad the inputs in W dimension as before so that all indirections corresponding to $r = 0$ fetch cacheline aligned vector of inputs. Second, for indirections with $r \neq 0$, we fetch a vector of inputs assuming $r = 0$ and left shift it by r. This means that the rightmost $R - 1$ values in the vector can not be used to accumulate the innermost activation group at those output positions. We mask the vector lanes corresponding to those positions and suffer less throughput. In general, we loose $V_w/(R-1)\%$ throughput in order to avoid split

loads where V_w is the vector width and R is the width of the filter.

This method requires prior knowledge about indirections to know the amount of left shift we need to do on the input vector. This can be done offline while populating the indirection tables. We sort the entries of `iiT` in increasing order of r in the innermost activation group. Consequently, we split `ccT` into R `ccT`s corresponding to each column of the filter. We read the `ccT` entries in increasing order of r to get the size of innermost activation group for that r. We then load and accumulate the innermost activation group as before according to the size returned by `ccT` entry. This results in R partially accumulated innermost activation groups. We appropriately shift them by corresponding r value once after accumulating all inputs and finally perform a masked vector add to get the final accumulated innermost activation group. Conceptually, we break the fast and medium fast path into R paths and the rest of the kernel, i.e., the slow path, remains the same.

However we find through experiments (Chapter 5) that step 2 doesn't give expected benefit because of less throughput of ALUs, more branch overhead due to multiple fast paths and extra shift operations. For these reasons we stop at step 1 after padding the inputs and suffer split loads for non 1×1 filters. We emphasize that this is a source of potential improvement and leave it for future work.

Input replication to remove split loads. Since the above described method works on less than ideal throughput of the functional units, it can hurt performance of layers where filters have large spatial dimension. Another way to handle split loads for non $1 \times s$ filters can be to replicate the input R times with each of the R replications such that it is cacheline aligned for the indirections in that column. However, this has a significant storage overhead so we do not employ this.

4.7 TILING AND DATAFLOW

Most of the modern CNN layers require a storage capacity that exceed the typical on chip L1 memory present in CPUs. Although L2 cache is typically much bigger in size than L1 (for example, 1MB L2 vs 32KB L1 in Skylake-X), it is still not enough to hold all three data types completely. This results in capacity misses and the functional units can get stalled waiting on the data to arrive from memory. So we tile the data accordingly to fit the working set at appropriate level in memory hierarchy. However, since tiling the data results in less working set size, the number of input channels in the working set might decrease. This will decrease the amount of repetition and consequently the performance.

It becomes important to choose an appropriate tile size based on layer parameters to get as much repetition as possible without suffering from cache misses. We do a careful analysis

Algorithm 4.5: Tiled UCNN Kernel with G=2 and output stationary dataflow

```
1  C_iter = ceil(C/C_t);
2  H_iter = ceil(H/H_t);
3  W_iter = ceil(W/W_t);
4  K_iter = ceil((K/G)/K_t) input[C_iter][H_iter][W_iter][C_t][H_t][W_t];
5  iiT[C_iter][K/G][C_t * S * R];
6  wiT_0[C_iter][K/G][C_t * S * R];
7  wiT_1[C_iter][K/G][C_t * S * R];
8  sccT[C_iter][K/G];
9  output[K_iter][H_iter][W_iter][K_t][H_t - S + 1][W_t - R + 1];
10 for kt: 1 → K_iter do
11     for ct: 1 → C_iter do
12         for ht: 1 → H_iter do
13             for wt: 1 → W_iter do
14                 for k: 1 → K_t do
15                     for h: 1 → H_t do
16                         for w: 1→ W_t; w+= V_w do
17                             for i: 1 → R*S*C_t do
18                                 # AGs and PSUMs are vector
19                                 AG_0 = AG_1 = 0;
20                                 PSUM_0 = PSUM_1 = 0;
21                                 IAG_size = ccT[k].pop();
22                                 FAST_PATH();
23                                 MEDIUM_FAST_PATH();
24                                 SLOW_PATH();
25                             end
26                             vstore(output[kt][ht][wt][k*G+0][h][w], PSUM_0);
27                             vstore(output[kt][ht][wt][k*G+1][h][w], PSUM_1);
28                         end
29                     end
30                 end
31             end
32         end
33     end
34 end
```

of several different layer types in modern CNNs and tile them at either L1, L2 or both.

The tiled dataflow is shown in algorithm 4.5. First, we tile the spatial dimension according to the vector width and spatial unrolling factor. This means that W_t and H_t are always chosen to be the vector width and unrolling factor respectively. This spatial tile ensures that the minimum required spatial dimension is present at L1 to fit as many input channels

as possible. We follow an output stationary dataflow at the L1 where we accumulate the pusms corresponding to one indirection table (or G output channels) with C_t number of input channels. We then iterate over this $W_t \times H_t \times C_t$ input (or logical L1 input tile) for K_t number of indirection tables to get reuse of this input tile.

We choose C_t and K_t such that the L1 input tile and K_t indirections fit in the L2. This allows us to iterate over the rest of the spatial input to get filter reuse over these K_t indirection tables. We then finish the complete output for each spatial position by iterating over all C_t tiles before finishing all output channels.

As mentioned before, it is important to hold a particular data type at either L1 or L2 to avoid cache misses and exploit the most reuse and repetition. Since the number of input channels affect both inputs and indirection tables, it becomes crucial to choose C_t wisely. Consequently, we divide the layers into two categories and choose C_t accordingly.

1× 1 layers. Since repetition depends on the filter volume, 1×1 layers will have the least amount of repetition for same number of channels, G and unique weights. For this reason, tiling at L1 turns out to be performance degrading even if most of the input loads hit in L1 purely because there isn't enough repetition for activation group reuse to work. We instead take the entire C into account and tile at the L2, i.e., C_t is chosen to be entire C so that a logical L1 input tile fits in L2 (or in L1 if C is small).

Non 1× 1 layers. Non 1×1 layers have significantly greater number of repetition that their 1×1 counterpart. For this reason, we can have lesser number of channels in them to get the same amount of repetition. This allows us to tile the inputs at L1 by choosing a $C_t \leq C$ which significantly reduces the number of misses that input loads suffer at L1. Given that non 1×1 layers will have a problem of split loads (Section 4.6), having a logical L1 input tile fitting in L1 helps undermining their effect and we still get good performance from the large repetition window.

4.8 EFFICIENT INDIRECTION ENTRIES

For a filter with dimension $R \times S \times C$, output at a spatial position (w, h) can be calculated as

$$output[(w,h)] = \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \sum_{c=0}^{C} input[(w+r, h+s, c)] \times filter[(r,s,c)]$$

This means that we need to fetch an input corresponding to the offset of each weight in the filter. For inputs stored in row-major format, this offset into the input buffer can be

calculated using the following equation.

$$offset(r, s, c) = w + r + (s + h) * W + c * H * W$$

where, (W, H) is the spatial dimension of inputs. This offset can be added to the base address of the input memory to get the actual address to fetch input from.

Trivially, we can store the offset of weight in the filter in the `iiT`, extract r, s and c from it in the kernel and calculate the offset into the input buffer. This would require multiple costly division operation which would quickly become a performance bottleneck. Instead, we separate out the above offset into two parts. One is dependant on the output spatial position $(w + h * W)$ and other depends on offset of weight into the filter buffer $(r + s * W + c * W * H)$. This allows us to compute the output spatial position offset once before traversing the indirection tables and perform an add with the `iiT` entry to get the actual offset into the input memory. This is very efficient at the expense of slightly higher number of bits to store each `iiT` entry.

CHAPTER 5: EVALUATION

5.1 METHODOLOGY

5.1.1 Measurement Setup

We evaluate our kernel on a 6-core Intel Skylake-X machine with AVX-512 enabled. We disable hyperthreading to avoid contention of multiple threads on same core and so that each thread gets full resources of a core. We also enable huge pages (2MB)[1] on the machine to avoid expensive TLB misses.

5.1.2 Point of Comparison

We evaluate the following design variants:

1. **MKL_DNN:** MKL_DNN [39] is an industry standard Math Kernel Library by Intel for Deep Neural Networks. We consider it the baseline which perform a GEMM based dot-product and does not exploit any type of sparsity or weight repetition. We measure the performance by actual wall-clock runtime. Since there can be variance in wall clock time across multiple runs, we take an averaged time across 100 runs for comparison.

2. **UCNN_opt:** Theoretical optimal UCNN kernel which performs the minimum number of operations required to compute the dot-product. This assumes that there are no software overhead of the algorithm and it maps perfectly to hardware in ideal conditions, i.e., all memory operations hit in L1, perfect throughput of ALUs. UCNN_opt gives a theoretical maximum performance that can be achieved using UCNN.

3. **UCNN:** UCNN kernel with all optimizations enabled. Like MKL_DNN we measure performance by averaging the wall-clock runtime of 100 inference runs.

5.1.3 CNN evaluated

We evaluate on ResNet-50 [12] which is a state-of-the-art network to for image classification task. We refer to it as ResNet for simplicity.

[1] HUGETLB_MORECORE=yes, LD_PRELOAD=libhugetlbfs.so

5.2 PERFORMACE ANALYSIS

Single Thread Performance. Figure 5.1 shows the relative improvement of UCNN_opt and UCNN over MKL_DNN for $U = 2$ and 3 for a complete inference of ResNet. As expected, UCNN for $U = 2$ (1.51×) performs significantly better than $U = 3$ (1.18×). An interesting observation is that there is significant gap between UCNN_opt and UCNN, indicating that there is a potential for further improvement. We emphasize that this gap can be narrowed down by smartly fixing the split loads problem (Section 4.6).

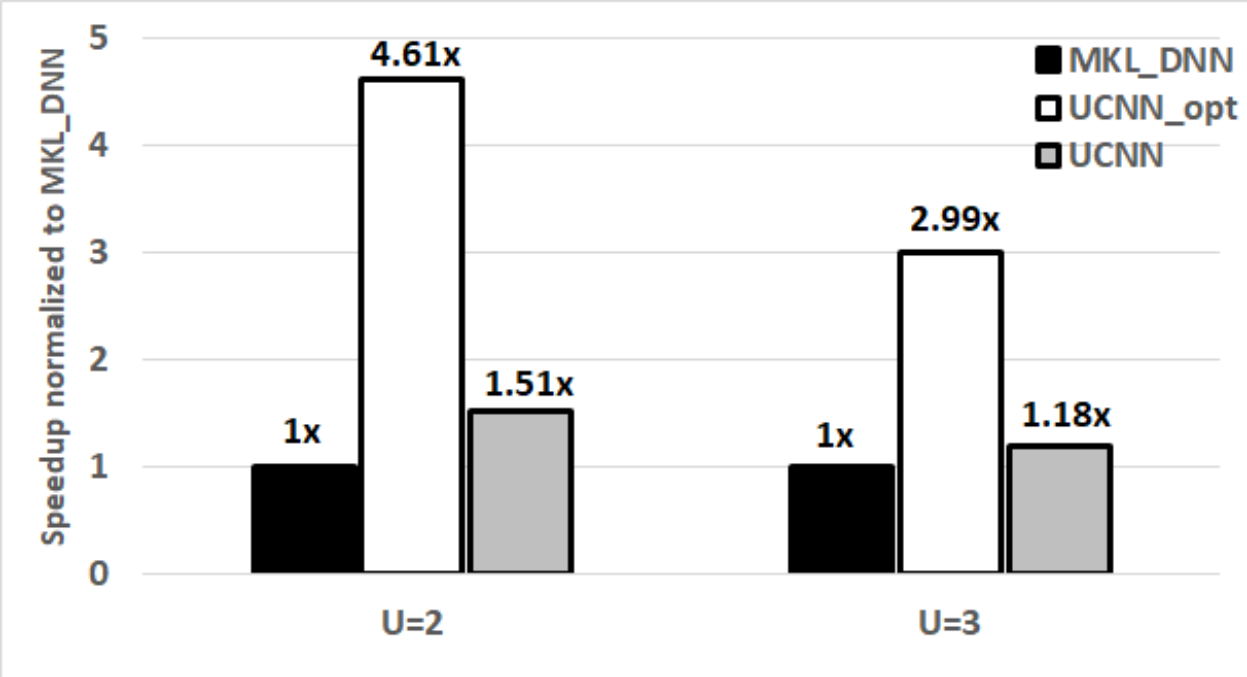

Figure 5.1: Single thread performance of MKL_DNN, UCNN_opt and UCNN on ResNet-50 relative to MKL_DNN.

Multithread Performance. We use OpenMP[2] to parallelize UCNN kernel into 6 parallel threads. To avoid contention between parallel threads, we choose to parallelize different output channels (Statement 10 in Algorithm 4.5).

Figure 5.2 shows the relative performance of UCNN_opt and UCNN compared to MKL_DNN for 6 threads. We see that UCNN kernel scales almost linearly with the number of threads by completely utilizing the inherent parallelization present in multiple output channels of CNNs.

[2]OMP_NUM_THREADS=6

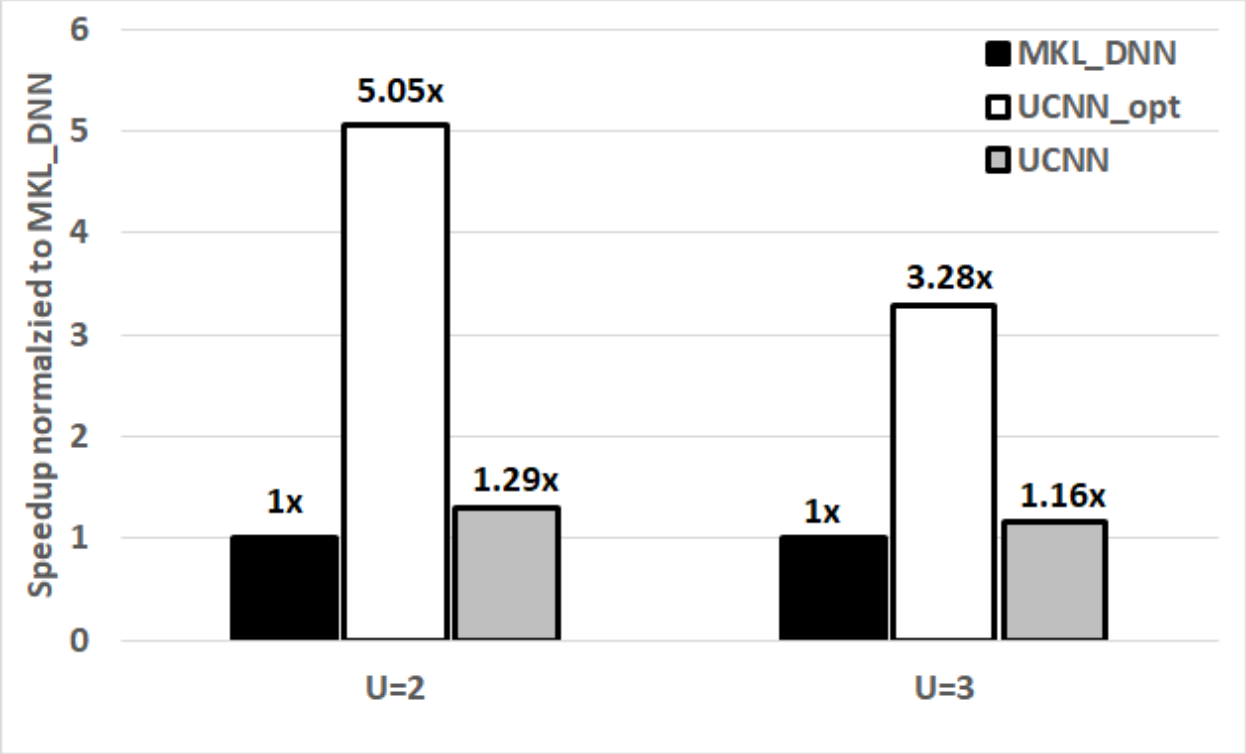

Figure 5.2: Multithreaded (6 threads) performance of MKL_DNN, UCNN_opt and UCNN on ResNet-50 relative to MKL_DNN.

5.3 MODEL COMPRESSION

Since UCNN stores indirections corresponding to multiple filters in compressed form (Section 3.2), it achieves significant level of compression over dense storage. Figure 5.3 shows the compression ratio of UCNN over Dense storage for different values of G. UCNN stores the weight indices instead of actual weight which enables a 1.7× compression over dense storage even for $G = 1$. This compression factor increases as G increases but reaches a saturation point at $G = 8$ because of large number if intermediate partial psums at the L1. For $G = 2, 3, 4$, which is usually enabled by $U = 2, 3$, UCNN can achieve close to 2× compression. We emphasize that UCNN greatly reduce the energy consumption for inference as significantly less parameters will have to fetched from DRAM but leave this for future work.

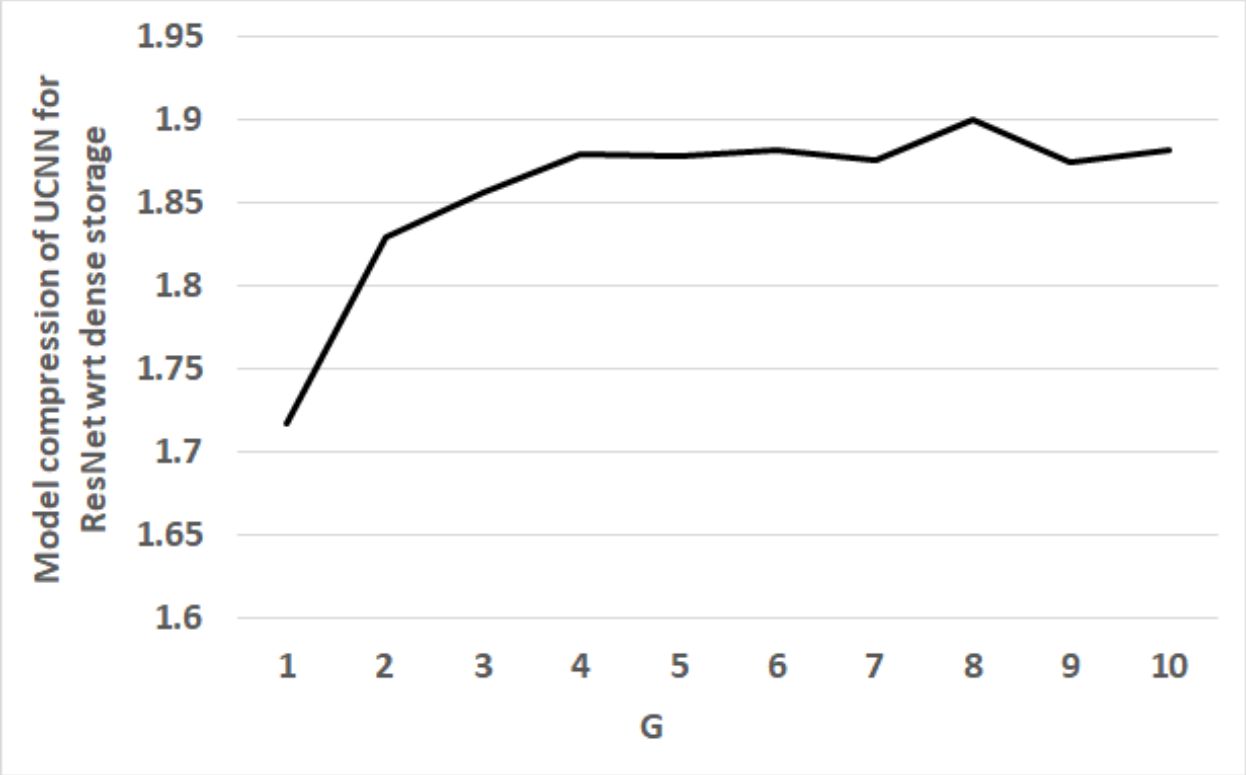

Figure 5.3: Compression achieved by UCNN over Dense storage of Inputs, Filters and Outputs for different G values. Sweep of G is performed for $C_t = 32$, spatial unrolling of 8 and $U = 2$ for AVX-512.

CHAPTER 6: CONCLUSION

This thesis proposed a highly optimized software kernel called UCNN that exploits weight repetition in modern CNNs to improve performance of inference on general purpose CPUs enabled with SIMD extensions. It shows that the irregularity introduced by weight repetition can be countered in a hardware aware software without any modifications to existing hardware. When compared to MKL_DNN, it achieves upto 1.51× improvement in runtime of inference on ResNet-50. We see our work as the first one to exploit repetition of weights on general purpose and programmable hardware and a generalization of sparsity.

REFERENCES

[1] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury, and T. Sainath, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, November 2012. [Online]. Available: https://www.microsoft.com/en-us/research/publication/deep-neural-networks-for-acoustic-modeling-in-speech-recognition/

[2] D. Ciregan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," ser. CVPR'12.

[3] J. Morajda, "Neural networks and their economic applications," in *Artificial intelligence and security in computing systems*. Springer, 2003, pp. 53–62.

[4] J. L. Patel and R. K. Goyal, "Applications of artificial neural networks in medical science," *Current clinical pharmacology*, vol. 2, no. 3, pp. 217–226, 2007.

[5] H. Malmgren, M. Borga, and L. Niklasson, *Artificial Neural Networks in Medicine and Biology: Proceedings of the ANNIMAB-1 Conference, Göteborg, Sweden, 13–16 May 2000*. Springer Science & Business Media, 2012.

[6] K. C. D. C. S. C. M. D. K. H. E. I. Y. J. B. J. T. L. H. L. Y. L. V. P. B. R. F. S. A. T. X. W. Y. W. B. W. R. X. S. Y. P. Z. C. Wu, D. Brooks, "Machine learning at facebook: Understanding inference at the edge," in *HPCA'19 (to appear)*.

[7] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, N. Pieter, S. Misha, X. Liang, and X. Wang, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *HPCA'18*.

[8] "Amazon machine learning," https://aws.amazon.com/machine-learning/.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, ser. NIPS'12.

[10] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," ser. CVPR'15. [Online]. Available: http://arxiv.org/abs/1409.4842

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," ser. CVPR'16.

[13] K. Simonyan and A. Zisserman, "Two-stream convolutional networks for action recognition in videos," ser. NIPS'14.

[14] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," ser. ICLR'16.

[16] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," ser. NIPS'15.

[17] J. Park, S. R. Li, W. Wen, H. Li, Y. Chen, and P. Dubey, "Holistic sparsecnn: Forging the trident of accuracy, speed, and size," *CoRR*, vol. abs/1608.01409, 2016. [Online]. Available: http://arxiv.org/abs/1608.01409

[18] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, Ed. Morgan-Kaufmann, 1990, pp. 598–605. [Online]. Available: http://papers.nips.cc/paper/250-optimal-brain-damage.pdf

[19] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," ser. ISCA'16.

[20] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," ser. MICRO'16.

[21] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," ser. ISCA'17.

[22] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," ser. ISCA'16.

[23] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," ser. ISCA'16.

[24] S. Sharify, M. Mahmoud, A. D. Lascorz, M. Nikolic, and A. Moshovos, "Laconic deep learning computing," *CoRR*, vol. abs/1805.04513, 2018. [Online]. Available: http://arxiv.org/abs/1805.04513

[25] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *ISCA'17*.

[26] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," ser. NIPS'16.

[27] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," ser. ICLR'17.

[28] C. Zhu, S. Han, H. Mao, and W. J. Dally, "Trained ternary quantization," ser. ICLR'17.

[29] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," ser. ISCA'17.

[30] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis, and M. Horowitz, "Dnn dataflow choice is overrated," 2018.

[31] K. Hegde, R. Agrawal, Y. Yao, and C. Fletcher, "Morph: Flexible acceleration for 3d cnn-basedvideo understanding," in *MICRO'18*.

[32] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *ISCA'18*.

[33] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[34] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *ICANN'14*.

[35] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1. Citeseer, 2011, p. 4.

[36] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[37] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *online: http://www. cs. toronto. edu/kriz/cifar. html*, 2014.

[38] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *CVPR'09*.

[39] "Intel mkl-dnn," https://github.com/intel/mkl-dnn.